SEApp: Bringing Mandatory Access Control to Android Apps

Matthew Rossi*, Dario Facchinetti*, Enrico Bacis*, Marco Rosa+, Stefano Paraboschi*

*University of Bergamo  +SAP Security Research

Speakers
Matthew Rossi  matthew.rossi@unibg.it
Dario Facchinetti  dario.fad@gmail.com
Motivation

Retain control on which components access sensitive data, and limit the impact of internal vulnerabilities

- every component has complete access to the internal storage
- 3rd-party libraries may abuse app privileges
- large and complex components prone to bug are not easy to isolate
Idea

1. Separate components into different app processes
2. Regulate with SELinux the permissions at process level

A component falls into a process based on its `android:process` value in the manifest.
Each app provides a policy module. The policy module lists the security contexts associated with processes and files.

file_contexts

.*
dir/unclassified  u:object_r:pkg_name.unclassified_file:s0
dir/secret  u:object_r:pkg_name.secret_file:s0

seapp_contexts

user=_app seinfo=cert_id domain=pkg_name.unclassified name=pkg.name:unclassified
user=_app seinfo=cert_id domain=pkg_name.secret name=pkg.name:secret

sepolicy.cil

(block package_name
  (type secret)
  (call md_appdomain (secret))
  (typebounds untrusted_app secret)
  (allow secret cameraserver_service (service_manager (find)))
  ...
Policy language syntax

Declare types and assign them permissions

- `type` and `typeattribute` declare types and attributes
- `typeattributeset` populates attributes to improve policy conciseness
- `allow` grants permissions to types and attributes
- `call` adds a set of predefined policy statements and improve usability

Guide and enforce constraints

- `block` wraps the policy in a unique namespace
- `typebounds` bounds the permissions of a type to a parent type
Policy language constraints

To preserve the overall consistency of the SELinux policy, each policy module:

- **must not** change the system policy
- **must** have an impact only on processes and resources associated with the app itself

Policy constraints are enforced at **installation time** and **runtime**
Origin of types and attributes

Using the `block` statement we detect the origin of types and attributes

- **local**: defined within the policy module (ns equal to the package name)
- **global**: defined by the system

Origin determines valid `allow` and `typeattribute` set policy statements

For example, the use of types or attributes defined by other modules is prohibited
Constraining `allow` statements

**AllowSS** represents a direct modification of the system policy **[denied]**

```
(allow isolated_app app_data_file (file (open)))
```

**AllowAA** defines privileges internal to the app module **[permitted]**

```
(allow mydomain mytype (file (create getattr open read write)))
```

**AllowAS** when a local type needs to be granted a permission on a system type **[bounded]**

```
(allow mydomain activity_service (service_manager(find)))
~ (typebounds untrusted_app mydomain)
```

**AllowSA** regulates how system components access internal types **[denied]**

```
(allow gpuservice mytype (file (open)))
```
Macros

But we need to **ensure interoperability** with services crucial to the app lifecycle (e.g., Zygote).

So we introduce macros.

```
(call md_appdomain (mydomain))
```

Benefits:

- safe by design, a predefined set of statements is added to the policy (i.e., by extending system domains that already have permissions on `untrusted_app` and `app_data_file`)
- no need for the app developer to know or understand system policy internals
Install time support

Changes to the app installation procedure
Runtime support: processes

Zygote was modified to enable runtime process labeling
Runtime support: files

Introduction of `restorecon` service to enable file labeling

The call to the `restorecon` service is performed transparently by `android.os.File`, a new API with the same interface of `java.io.File`
Boot procedure

Since Treble:

- policy segment changes → on-device compilation

Changed the second stage (init.rc):

- to mount /data partitions (where policy modules are stored)
- implementing a new built-in function to build and reload the policy

The policy is not bypassable, since the modules are loaded before any application starts
Experiments: install time overhead

Device: Pixel 3, Version: android-10.0.0_r41
## Experiments: runtime overhead

<table>
<thead>
<tr>
<th>Component</th>
<th>Cold start (ms)</th>
<th>Warm start (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\mu)</td>
<td>(\sigma)</td>
</tr>
<tr>
<td><strong>Stock OS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LocalActivity</td>
<td>39.102</td>
<td>1.094</td>
</tr>
<tr>
<td>SEAppActivity</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>SEApp</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LocalService</td>
<td>19.164</td>
<td>1.444</td>
</tr>
<tr>
<td>RemoteService</td>
<td>105.467</td>
<td>2.800</td>
</tr>
<tr>
<td>IsolatedService</td>
<td>103.923</td>
<td>2.425</td>
</tr>
<tr>
<td>SEAppService</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### File creation

<table>
<thead>
<tr>
<th>Test</th>
<th>(\mu) (\mu)</th>
<th>(\sigma) (\mu)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock OS</td>
<td>57.077</td>
<td>5.174</td>
</tr>
<tr>
<td>SEApp</td>
<td>60.696</td>
<td>6.782</td>
</tr>
<tr>
<td>SEApp + restorecon</td>
<td>431.472</td>
<td>109.494</td>
</tr>
</tbody>
</table>

Device: Pixel 3, Version: android-10.0.0_r41
Thank you! Any questions?

**SEApp**

🔗 [https://github.com/matthewrossi/seapp](https://github.com/matthewrossi/seapp)
Available for Android 10, 9
Tested on Pixel 3, Pixel 2 XL, Emulator

**Our contacts**

Matthew Rossi  
Dario Facchinetti  
Enrico Bacis  
Marco Rosa  
Stefano Paraboschi

Matthew.rossi@unibg.it  
dario.fad@gmail.com  
enrico.bacis@gmail.com  
marco.rosa@sap.com  
parabosc@unibg.it